CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

User Manual for KRAUT — The Interim Spice Debugger

Bernd Bruegge

10 Feb 84

Abstract

This manual describes version 5.0 of KRAUT, the debugging system for Perq Pascal running on the Perq Systems PERQ computer under the Accent operating system. KRAUT provides most of the commands of traditional symbolic debuggers such as setting of breakpoints, state inspection/modification and source file access. It also contains MACE, a low level debugger for the inspection of the target process on the qcode and microcode level. The novel feature of KRAUT is *Path Rules*, a flexible debugging mechanism based on path expressions.

Spice Document S156

Keywords and index categories:

Location of machine-readable file:

/usr/spicedoc/manual/spiceprogram/kraut/kerman.mint@CMU-CS-Spice

KRAUT is a contribution of Siemens RTL, Princeton, New Jersey to the SPICE project. This document is a revised version of the technical report RTL-83TR-008 from Siemens RTL.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

Table of Contents

Chapter One
Introduction

KRAUT is a remote symbolic debugger for Perq Pascal running under the Accent operating system. One of the novel features of KRAUT is that the user can specify the expected behaviour of a computation with path expressions. The debugger runs in its own address space and inspection and modification of the target process address space is done by means of Accent kernel calls. Currently both the debugger and the target process have to reside on the same Perq, but later it will be possible to debug processes which are located on a physically remote Perq.

The manual is organized as follows: Section 1.1 explains how to retrieve the newest version of the debugger, how to generate symbolic information and how to invoke the debugger. Pascal's scope rules have been extended in KRAUT and are described in Section 2. The major design objective of KRAUT was to test the viability of path expressions in the context of debugging. Path expressions are embedded in path rules which are introduced in Section 3. In addition to path rule commands, KRAUT provides traditional debugging commands for setting breakpoints, inspecting and modifying variables in the user process address space, opening source files and searching for text strings. The complete command language of KRAUT is described in Section 4. KRAUT contains the low level debugger MACE, which allows the inspection and modification of the target process on qcode and (partially) on microcode level. MACE is intended mainly for compiler writers and microprogrammers and is described in Section 5. KRAUT is still actively being developed and internal debugger errors might show up while you are debugging your program. Section 6 shows you how to cope with these errors and Section 7 mentions the current restrictions and known bugs of the debugger.

1.1 Getting Started

If KRAUT is not yet part of your Accent system or if you want to get a newer version, you have to retrieve it from the CFS Vax@CMU. KRAUT can be retrieved using the `update` program with the following command:

```
update krautrun/test
```

The run file is `debugger.run` and it has to reside in the partition `<boot>`.

Symbolic information for a module is produced by the compiler if the module is compiled with the `/scrounge` switch. For example

```
compile test.pas/scrounge
```

generates the symbolic information for `test.pas`. The `scrounge` switch is on by default. If you want to suppress the generation of the symbolic information use `noscrounge`. The symbolic information is contained in two files with the extension `sym` and `qmap`[1]. KRAUT must have access to these files to allow symbolic debugging at other than a routine and module name level.

KRAUT can be invoked in the following ways.

1. It can be called on the shell level by closing the run command with the ↑ character: for example **RUN test↑**. This allows the definition of path rules, debugger variables and breakpoints *before* the execution of the program.
2. Typing the Shell command **DEBUG**[2] will suspend the execution of a running process. In this case the **Spice Process Manager** reports a `Trap 0` in the target process and invokes the debugger. This allows you to catch processes in dead loops or in I/O wait states.
3. Any uncaught exception in the target process will invoke KRAUT.
4. KRAUT can be invoked via SAPPHIRE if the item **Debug Process** is selected from the Pop-Up Menu. The menu is displayed from a window or an icon in a manner that is consistent with regular SAPPHIRE window management.

In all of the above methods of invoking KRAUT, a bug symbol appears in the icon corresponding to the window running the process in question. The SAPPHIRE cursor appears on the screen and prompts for the creation of a Debug Window.

KRAUT tries to set the current directory to the directory where the main program was compiled. If that directory does not exist, a warning is given and the current directory is left as <current>.

KRAUT does a consistency check between symbolic information and object files: if a symbol table was not produced at the same time as the object file or if the source file is younger than the seg file a warning is issued. In this case the debugger should be used with caution, because *any* information from those modules could be wrong. Any information retrieved from inconsistent symbol tables is marked with a '?'.

KRAUT has a transcript facility to document errors that cannot be reproduced and bugs in modules not written by yourself. If you encounter a bug you cannot cope with, you might want to send the transcript file to the maintainer of the module (see **MAINTAINER** and **REPORT** commands). Transcript files are generated by default, but you can turn the transcript switch ON or OFF (see **SCRIPT** command). If the transcript switch is turned OFF in the default command file the generation of a transcript file is suppressed.

---

1 The `sym` file contains the names of the variables and routines declared in the program. The `qmap` file contains a mapping between qcode offsets in the seg file and the corresponding source statements.

2 **DEBUG** expects the name or the number of the target process as parameter, which can be found with the shell command **SYS**.

A transcript file has the name "M.kscr.i", where M is the extensionless name of the file containing the main program, and where i = A,...,Z. If the name space of the transcript files is exhausted, you are prompted for a file name.

At the beginning KRAUT prints out a stack trace of the last 4 routine calls. If the symbol table information files exist, calls are written out in terms of the source program. If no symbolic information is available, the low level debugger MACE writes out the routine call in terms of the target code. For example

```
| Uncaught exception: Division by zero
|  [EXCEPT''RAISEP]  Q321(19,4,2306,2306, ...)
|  DODIV (23 test.pas;1)  j := a div parm;
|  EXECUTE (17 sys>user>bob>bug.pas;1 )  Dodiv(a,parm);
|  ? MAIN (34 test.pas;1) parm := 0; Execute(parm);
```

shows the trace of a program that did not catch the exception `Division by Zero`. The source line for the routine on top of the stack could not be found, because the defining module was compiled with the `noscrounge` switch. The next three lines are given in terms of the source text. The last source line is marked with a '?' indicating that the symbolic information was not generated at the same time as the seg file. If the top routine is a predefined exception, the current routine is automatically set to the routine below the top routine, otherwise the current routine is set to the top routine.

KRAUT looks for 2 files[3] whose names are derived from the extensionless file name `M` containing the main program declaration. The file `M.switches` defines certain user definable constants and switches such as the prompt sign and whether a transcript file of the debugging dialogue is to be generated. The `M.kmd` file must be a KRAUT command file and its commands are executed before any other command can be typed in by the user. If `M.switches` does not exist the debugger assumes the following defaults: The prompt sign is "|" and the transcript switch is ON. If `M.kmd` does not exist, control is immediately handed over to the user.

[3] File lookup is always done with Accent's file search list.

Chapter Two
Naming

Since KRAUT is a symbolic debugger it should be able to access any object mentioned in the source program. However, in many cases the current symbol table format does not provide enough information to access the full name space or to enforce Pascal's type rules. This section describes several extensions made to the Pascal syntax to allow symbolic debugging despite these constraints.

2.1 Constants

A constant can be prefixed by a '#', a base indicator and a type indicator. The base is indicated by B, D, H or O (binary, decimal, hexadecimal or octal) and the type by L or I (long or integer). If the base or type indicator is omitted, the defaults D (decimal) or I (16 bit constant) are assumed. A normal Pascal constant in the range -32768 to + 32767 is of type integer, otherwise of type long.

Examples:

`#OL4743337`	`32 bit octal long`
`#HLABF9DC93`	`32 bit hexadecimal long`
`#b0101110101111100`	`16 bit binary`
`#3434`	`16 bit decimal integer`
`3434`	`16 bit decimal integer`
`986966`	`32 bit decimal long`

2.2 Types

The debugger tries to do some type checking and issues a warning if it finds a violation of Pascal's type rules. However, only simple types are checked correctly. *Arrays* are of the same type if they have the same subtype. *Records* are considered equivalent if they have the same length. All *pointers* are of the same type.

The type of any variable or Pascal expression can be explicitly specified by one of the following type qualifiers:

:array[n]	Array
:boolean	Byte
:char	Char
:integer	Integer
:long	Long
:pointer	Pointer
:record[n]	Record
:real	Real
:string	String
:set[n]	Set

Each type qualifier is uniquely specified by the underlined characters. The optional expression [n] specifies the wordsize of the structure. If [n] is not given, the default [1] is assumed.

Examples:

```
Var
   a: array[0..10] of integer;

   rec: record
         a,b,c : integer;
         d : pointer;
        end;

   i,gi : integer;

   .......
```

a[j]:iong denotes a 32 bit variable.

*(rec.3↑:long + 5*gi:integer):s* is interpreted as string.

gi:int := #b1101011111 assigns the bit pattern 0000001101011111 to *gi*.

i:set[1]= prints out the value of *i* as 16 bits[4].

2.3 Scope Rules

KRAUT allows you to circumvent the scope rules of Perq Pascal. For example, variables from the ***private*** section of a module can be accessed, even if they are not visible. The Pascal notation for an identifier has been extended: To denote the variable Foo in routine P you can write P'Foo. And M''P'Foo denotes the variable Foo in routine P in module M. More generally, M''P_1'...'P_{n-1}'P_n,i denotes variable i in

[4] Note that the least significant bit is printed out first.

routine P_n nested in P_{n-1},..., nested in P_1, which is declared in module M[5]. One can also access local variables of routines currently allocated, but not visible. Local variables of activated routines that are outside the current static scope can be accessed by specifying the call chain in front of the variable name. A call chain consists of names of routines separated by a back slash ("\"). For example `M''P\M1''foo\M1''foo'i` denotes the variable `i` in the second invocation of routine `foo` in module `M1` called from routine `P` in module `M`.

Such a denotation may specify an object that is currently not in existence. If it is not part of a predicate (see section 3.1), the debugger prints an error message and returns to the user level. For predicates, a three-valued logic is assumed and predicates that cannot be evaluated will yield the predefined value ←UNDEFINED. Any composite predicate containing undefined values will also yield ←UNDEFINED.

Identifier search in KRAUT follows the following algorithm:

1. If the name does not contain a module or routine qualifier, then it is parsed according to Perq Pascal's scope rules: First the name is looked up in the runtime stack starting at the current routine. If this is not successful, then the name is looked up in the *current module*. The *current routine and module* can be set by the CURRENT command.
2. If the name is of the form `P'i`, KRAUT searches the runtime stack for each module mentioned in KRAUT's search list for a routine with name `P`. If such a routine is found, KRAUT looks for the local variable `i`. The symbol tables of the modules are searched in the order defined by KRAUT's *module search list* (see section 4.1).
3. If the name is of the form `M''i`, KRAUT looks for variable `i` in module `M`. Note that KRAUT does not distinguish, whether `i` is in the private or in the exported section.

---

5 A note concerning the *double quote*: It is not possible to use only one quote because of the possibility of name conflicts. For example, in the following program

```
program foo;
var i : integer;

procedure foo;
 var i : integer;
....
```

the name `foo'i` could either mean the global `i` in module `foo`, or the local variable `i` in routine `foo`.

Chapter Three
Path Rules

Originally path expressions were developed for the synchronization of concurrent processes. In KRAUT they are used as pattern recognizers to monitor the behaviour of a target process. Path expressions are embedded in the more general notion of a *path rule*. In this section we show how path rules are defined and give some examples for their use in debugging. A path rule consists of two parts: an event recognition part and an action part. The event recognition part consists of a *generalized path expression* which is discussed in section 3.1. The purpose of the action part is to describe what the debugger has to do in the case of a match or mismatch of the actual computation with the computation described by the recognition part. The action part is described in section 3.2.

3.1 Generalized Path Expressions

Generalized path expressions are basic path expressions extended by counters and predicates.

A *basic path expression* is a regular expression of operands connected with the operators repetition (*), sequencing (;) and exclusive selection (|). The operands are called *path functions* in the following. Any routine defined in the source program is a predefined path function. Other path functions can be defined during the debugging session with the **DEFINE** command described in section 4.

If a path function R is used in a path expression it matches either the activation or termination of the execution of R in the target process. Path functions can be postfixed with an *event qualifier* to specify either the activiation or termination of a routine: If R is a path function, then R!A denotes the activation of R and R!T denotes the termination of R. Thus R can be seen as a shorthand notation for the generalized path expression (R!A | R!T).

A *counter* can be defined explicitly with the **Counter** command described in section 4. Furthermore, there are two predefined counters for each path function mentioned in a path expression: The ←ACT counter describes how many times the path function has been activated and the ←TERM counter describes how many times it has been completed[6].

A *predicate* is a relational expression consisting of implicit or explicit counters and names from the name

---

6 If these counter names conflict with names of objects in the target process, they have to be prefixed with the escape character '←'.

space of the program. Predicates have to be enclosed in curly brackets and can be associated with any path function[7]. There is a predefined predicate ALWAYS which is equivalent to TRUE=TRUE.

An example of a generalized path expression is

```
InitStack;
(Push{(←TERM(Push) - ←TERM(Pop)) < N)}|
(Pop | Top){(←TERM(Push) - ←TERM(Pop) > 0};)*
```

which states the operational constraints on a bounded stack of length N: First the routine InitStack has to be called. One of the following can then happen: Either Push can be called as long as the size of the stack is smaller than N, or Top or Pop can be called if the size of the stack is larger than 0.

Generalized path expressions can be used in two different ways: If a generalized path expression is prefixed by the keyword Find, the specified execution sequence is matched against the observed execution sequence and the path action specifies what to do in the case of a match and a mismatch. Find expressions are useful in searching for the pattern of a particular execution sequence. For example,

```
FindPath BeginLoop
  WhileLoop{←ACT(Push)= N and ←ACT(Pop) = 1}
```

looks for the activation of the while loop when the routine Push has been called N times and Pop once.

The other use of generalized path expressions is to *enforce* a particular execution sequence. If a generalized path expression is prefixed by the keyword Check, the specified execution sequence is matched against the observed execution sequence. The path action contains commands about what to do in the case of a violation or nonviolation. For example,

```
CheckPath Loop
  (WhileLoop{←ACT(Whileloop) < 6} | Pop)*
```

says that the while loop should not be executed more than 6 times before a call to Pop occurs.

Any identifier that is used in a predicate of a generalized path expression is called a *path variable*. Predefined path variables are identifiers explicitly declared in the source program and accessible via the symbol table.

---

[7] If a path function P is mentioned without counters or predicates the defaults ←ACT(P) >= 0 and ←TERM(P) >= 0 are assumed.

3.2 Path Actions

A path action is declared by the keyword `Action` followed by the identifier of the path expression with which the path action is to be associated. The keywords `Match` and `NoMatch` are followed by the actions to be taken in case of a match and mismatch, respectively. Any sequence of debugger commands not containing an *assignment* or *routine call* is a valid action. Assignments and routine calls have to be preceded by a HALT command. If an action contains more than one command, the commands have to be separated by '~'. Either the `Match` or the `Mismatch` part or both can be omitted, in which case no action will be associated with the missing part.

For example

```
Action Loop
        Match   => Writeln("I = ", I) >> LoopWindow
        NoMatch  => HALT
```

associates the debugger commands HALT and `Writeln("I = ",I >>` LoopWindow) with the path expression `Loop`. Thus whenever the evaluation of `Loop` [8] yields a violation of the specified ordering the computation is suspended and control is turned over to the debugger, otherwise the value of I is written into the window `LoopWindow`.

The association of a path action with a generalized path expression is dynamic. Thus you can remove a path action from a path rule and replace it by another path action. This can be done any time the target process is suspended: If the process is still running, you can suspend it with the characters ↑DEL k or ↑DEL d (see page 19).

3.3 Manipulating Path Rules

Path rules are automatically enabled when they are defined. They can be disabled and enabled again by the DISABLE and ENABLE commands described in section 4.6. Note that if a `Match` or a `Mismatch` occurs, the path rule is not automatically disabled. Disabling a path rule and successive enabling has to be done explicitly. Every time the `Match` part of a path rule is executed the path rule is set to its initial state, that is, is starting over again to look for the pattern specified in the generalized path expression.

---

8 Note that the generalized path expression LOOP is only evaluated when one of its path functions is executed

Chapter Four
Command Language

Any command can be abbreviated as long as the abbreviation is unique. For convenience, an attempt has been made to allow two letter abbreviations for nearly all commands. Because of this reason some of the command names might look a little bit strange. Commands and variable names are not case sensitive. For example, 'R' and 'r' mean the same command or identifier. If more than one command is entered per line, the commands have to be separated by the character "~". A command or command sequence can be extended over several lines. Each of the lines of such a command has to be terminated with a '\' character. For example,

```
callstac\
k 1~writ\
eln('I = ',I)~\
go
```

is parsed as:

```
callstack 1~writeln('I = ',I)~go
```

At any time the debugger maintains a current line number and a current source file which can be denoted by the dot character (".").

4.1 Search List Commands

KRAUT maintains a search list of open modules which determines the order in which symbol tables are searched for identifiers. The search order is the reverse of the order in which the modules are entered: The module opened latest will be searched first. By modifying the search list appropriately you can speed up the debugger significantly. The module search list is initially empty. Any time the target process is suspended, the modules of routines on the runtime stack not yet in the module search list are appended to the tail of the module search list. Modules are never implicitly removed from the list.

Kraut looks up the source, qmap and sym files of a module using the current file search list. The SETSEARCH command can be used to add directories to the file search list. This makes it possible to access remote files from inside the debugger. If the file search list did not contain the directory in which the source or symbolic files are located, when the module was OPENed, KRAUT will try to open the module without this information. If you want to add theses files later, CLOSE the module, call SETSEARCH with the appropriate (remote) directory and then call OPEN again.

ChDir D

Set the current directory to D. If no argument is specified, show the current directory.

Open M

Insert module M at the head of KRAUT's module search list. If the module is already in the search list, it will be moved to the head of the list. The variable *Current File* is changed to the file name of the module opened. If M is a list of modules M_1 M_2 ...M_n, then open each of the modules as described above. If M = * , then open all the modules of the target process, except for the modules of the Pascal runtime system. The latter ones can be opened with the **System** command.

Close M

Delete module M from KRAUT's search list. If M is a list of modules M_1 M_2 ...M_n, then delete each of the modules. If M = * , then close all the modules of the target process.

SetSearch F_1',...,F_n

If no arguments are given, then show Accent's file search list. Otherwise modify the file search list as follows: If F_i is a directory name, push it on the search list, if F_i is a "-" then pop the top entry from the search list.

Show Modules M

If M is not given, show all open modules from KRAUT's module search list. Otherwise give some information about module M. This includes date of compilation, names of seg, qmap and sym files, name of imports, etc.

System

Initially the Pascal runtime system modules are not in the search list, even if they are explicitly imported by the user program. The Pascal runtime system consist of the module **PascalInit** and all its imports. **System** opens these modules and adds them to the search list.

4.2 Trace Commands

Traces are implemented internally as path rules. For each routine R to be traced, two path rules are created. The first one has the name ←AR and monitors the Activation of R. The second one has the name ←TR and monitors the Termination of R. The MATCH part in the path action is preset to print whether the routine is entered or exited. Traces can be edited with the ***EditPathRule*** command (see page 15).

Trace [Before |After] R [UserActions]

R can be either a module or a routine. If R is a module tracing path rules are created for each routine defined in the module. If it is a routine and neither `Before` nor `After` is specified, both path rules ←AR and ←TR are created. If `Before` is given, only the path rule ←AR to trace the entry of R is created. If `After` is given, only the path rule ←TR to trace the exit of R is created. If "UserActions" is specified, then "UserActions" are executed whenever one of the trace path rules fires.

DTrace [Before |After] R

If R is a routine delete the tracing path rules for R. If R is a module delete the traces for all routines of R. If `Before (After)` is given, delete the only the path rule for the entry (exit) of R.

4.3 Breakpoint Commands

Setting of breakpoints can be done either by specifying a source line or by a routine name. In the following, the symbol R denotes a routine. R can be of the form `M''P` or P. The symbol '#' denotes a source line in a source file in a certain directory. If the file name is omitted, the *current file* is assumed. The line number must correspond to a source line containing the beginning of a Perq Pascal statement, otherwise KRAUT writes an error message. Breakpoints are implemented internally as path rules. The name of the path rule is of the form ←B# [9], where # is in the range 1..150. The MATCH part in the path action is preset to the command sequence **Halt~Exception~Callstack 1**. Breakpoints can be edited with the ***EDITPATHRULE*** command (see page 15).

NOTE: The **Halt** command is part of the definition of a break point and cannot be removed from the **MATCH** part.

Break (after) R , Break #

Set Breakpoint at (after) routine R or at line #. For example `Break 4 main.pas` means: Set a break point at source line 4 in file main.pas.

---

9 Note that the character "←" is actually the character "_" on your keyboard

DBreak *

Delete all Breakpoints.

DBreak (after) R, DBreak #

Delete Breakpoint at (after) routine R or line #.

Show Breaks

Show current breakpoint list. This command is the same as **Show Pathrules** if no other pathrules but breakpoints are active.

4.4 Editor Commands

The following commands allow you to inspect source file and search for string patterns. If a new file name is specified, the *current file* will be set to the new name.

Type i j F

Show source lines **i** to **j** in file **F**. The *current source line* is set to the last line displayed. **F** must be visible with Accent's file search list. There are several abbreviations of the type command: The keyword **Type** can be omitted. If the file name is also omitted, then the current file is assumed. Furthermore, the commands **i F** or **'i** will display line **i** in file **F** or in the current file, respectively.

Around i F

Show 20 source lines around line **i** in file **F**. Line **i** is indicated by '***'. If **F** is omitted, the current file is assumed. If **i** and **F** are omitted the current source line is assumed.

Scroll i F

Type 20 source lines starting at the current source line. The **Next** command can be used after the first scroll.

Search 'S'? F

Search backward in file **F** for string **S**. If **S** is omitted take the string from the last search command. If **S** contains a "'", then two "'" have to be specified. Note that the search is not case sensitive. If **F** is omitted, the current file is assumed. If 'S' is found, the *current source line* is set to the line containing 'S'.

Search 'S'! F

Search forward in file F for string S. The '!' can be omitted if S is given. If S is omitted take the string from the last search command. If S contains a "'", then two "'" have to be specified. Note that the search is not case sensitive. If F is omitted, the current file is assumed. If 'S' is found, the *current source line* is set to the line containing 'S'.

Grep 'S' F

Look for all occurences of string S in file F. The search is case insensitive. If F is omitted than do string search starting at the current source line in the current source file. If 'S' is replaced by !, then look with the last string pattern specified. Currently neither S nor F can contain wildcards.

4.5 Definition Commands

KRAUT allows the definition of new path functions, counters and path variables which can be used in generalized path expressions in the same way as that for predefined names.

Counter C := I

Declare a counter variable C and assign it the value I. If the assignment is omitted, the counter is initialized to 0. Counters are regarded as global variables defined in a static scope surrounding the main program. If there is a variable with the same name defined in the program, KRAUT prefixes the counter variable with the escape character '←' and asks you for confirmation.

DCounter C

Delete the counter variable C.

Define Function P B E

Associate the path function P with a group of statements indicated by the source lines B and E in file F. B and E must belong to the same source file and the source line denoted by B must be smaller than the source line denoted by E.

4.6 Path Rule Commands

The debugger maintains a list of path rules which can be defined and manipulated. In the following, R denotes the name of a path rule and can be any identifier.

Action R A

Bind action A to path rule R. The arguments R and A can be omitted, in which case you are prompted for them. R must have been defined before by a `FindPath` or `CheckPath` command. A is of the form:

MATCH *Cmd1* NOMATCH *Cmd2*

Cmd1 and *Cmd2* can be any KRAUT command or command sequence. The `Match` part has to be defined before the `NOMATCH` part. Command sequences must be enclosed in parentheses. If the execution sequence matches the sequence specified by the generalized path expression, then *Cmd1* is executed. If there is no such match, then *Cmd2* is executed. For example

```
Action R1 MATCH (calls~br firstcall~disable R1~enable R2)
```

executes the **Calls** command, sets a breakpoint at routine `firstcall`, disables itself and enables path rule R2.

CheckPath R G

Qualify R as a Check rule and add it to the active path rule list. G is a generalized path expression (see section 3.1). If the arguments R and G are omitted, you are prompted for them and for the associated path action.

Disable R

Disable the currently active path rule R. This command takes the path rule R from the active path rule list and appends it to the list of inactive path rules.

Editpathrule P

This command allows to change the components *Generalized Path Expression*, *Match* and *NoMatch* of path rule P. Furthermore the user is prompted for the setting of two switches: The ENABLED switch enables P if set to true, otherwise disables it. If the VERBOSE switch is set to true, the interpretation of the path actions is done verbose, otherwise quiet. *Note that editing of path rules created by the* `BREAK` *command is restricted in the following way: It is not possible to change the whole Generalized Path Expression associated with the breakpoint, but only its predicate. Predicates can be entered when the CONDITION prompt is given.*

The following example sets a breakpoint at line 86 and changes the default action to do the following commands: Print out the callstack to depth 4. print the locals of the current routine, write the value of i and continue. The action will only occur if the target process executes line 86 and i has the value 1000.

```
|BREAK 86 test3.pas
|Set Breakpoint ←B1 at line 86 in TEST.PAS: write('First;');
|
|EDIT ←B1
|    CONDITION [] : i = 1000
|    MATCH [] :  Callstack 4~locals~writeln('i = ', i)~go
|    NOMATCH [] :
|    ENABLE [] : TRUE
|    VERBOSE [] : TRUE
```

Enable R

Enable the currently inactive path rule R. This command takes the path rule R from the inactive path rule list and appends it to the list of active path rules.

FindPath R G

Qualify R as a Check rule and add it to the active path rule list. G is a generalized path expression(see section 3.1). If the arguments R and G are omitted, you are prompted for them and for the associated path action.

Show PathRules (enabled | disabled)

Show the active or inactive path rule list. If no argument is specified show the entire path rule list. (This command is equivalent to **Show PathRules**).

Show PathRules F

Show the components of path rule F.

RemovePath P

Delete path rule P from the path rule list. If P is a list of path rules P_1 P_2 ... P_n then remove each of these path rules from the list.

Remove Action R

Delete the action bound to path rule R.

4.7 Variable Inspection Commands

In the following V and Id denote identifiers as defined in section 2, page 4, counters as defined in section 4.5, page 14 or history variables. Id can also be an arbitrary Pascal expression with the restrictions in section 7, page 37.

Globals M

Display all globals of module **M**. If **M** is not specified, display the locals of the *current module.*

Locals R

Display all locals of routine **R**. If **R** is not specified, display the locals of the *current routine.*

Parameters R

Show the current values of the parameters of R. If R is not specified, then show the parameters of the *current routine.*

Radix N

Set the current radix to N where N is a decimal number from the set { 2,8,10,16}. Initially the current radix is 10.

V=

Get the value of V following the scope rules as described in section 2.3. If the name of the variable is not also a KRAUT command, then the equal sign (=) can be omitted. To avoid confusion arising from names used in more than one module, values are always written out in the form "***value [Module]***". ***Module*** is the name of the module containing V or it is the string **Internal Counter** if V has been defined as a counter variable. If the value was retrieved from an inconsistent symbol table, the module name is marked with a '?' sign.

V := Id

Assign the value of Id to V following the scope rules described in section 2.3. V cannot be a history variable. The debugger tries to do some kind of type checking as described in section 2 and issues a warning if there is a violation.

Write(p1,p2,...,pn), Writeln(p1,p2,...,pn)

Write and **Writeln** are similar to Pascal's write and writeln. The only difference is that the output is always written in the current window. $P_1,P_2,...,P_n$ can be any Pascal expressions with the restrictions and extensions described in section 2, page 4. For example, if **foo = 'df'** and **bas = 1**, then **Writeln('FOO = ', foo, 'BAS = ', bas)** results in the output:

FOO = 'df'BAS = 1

4.8 Next Command

<CR>

This command is interpreted in the context of the last command. The next command is implemented for the following commands:

$	Mace command: Display the next location in the current mode and current radix.
Around	Show the next 20 source lines in the current file.
Calls	Display one more call. The current source line and file are updated to the line displayed, so typing `Around` after this command displays the context around the source line. Note: The current routine is not changed.
DownStack	Move down one activation record in the dynamic call chain.
Forward Search	Continue the search with the previous argument starting at the current source line.
Scroll	Show the next 20 source lines in the current file.
SingleStep	Execute one more source line.
Type	Show the next 20 source lines in the current file.
UpStack	Move up one activation record in the dynamic call chain.

4.9 Runtime Stack Commands

BottomStack

Set the *current routine* to the routine described by the activation record at the bottom of the runtime stack.

Calls N

Display the dynamic calling sequence of the target process. N specifies the number of calls to be printed. If N is omitted, then the whole sequence will be printed.

DownStack N

Move down one activation record towards the bottom of the runtime stack and update the *current routine*. If `N` is omitted, `N` = 1 is assumed.

TopStack

Set the *current routine* to the routine described by the activation record at the top of the runtime stack.

UpStack N

Move up N activation records towards the top of the runtime stack and update the *current routine*. If `N` is omitted, `N` = 1 is assumed.

4.10 Target Process Control Commands

These commands allow to modify or observe the execution of the target process.

Process Control Characters

The process control characters ↑DEL c, ↑DEL d, ↑DEL k and ↑DEL t are intercepted by KRAUT. Their interpretation depends on the context of the last command:

EXECUTE,GO

↑DEL t: Show the state of the target process. (Not yet implemented)

↑DEL c, ↑DEL d, ↑DEL k: Suspend the target process and return to command level.

↑Del r: NOOP

All other commands

↑DEL t: Show the state of the target process. (Not yet implemented)

↑DEL r,↑DEL d: Abort the current command.

↑DEL c: NOOP

Execute $R(p_1,p_2,....,p_n)$

Execute routine R with parameters $p_1,p_2,....,p_n$. R can be the main program, any global routine or a nested routine statically visible from the current point of execution. If R is the main program, you are prompted whether the run time stack shall be cleared or not. If R has been defined with parameters, they have to be passed, but the debugger does no type checking at all. If the closing parenthesis is omitted, the debugger writes out the type information for each parameter and asks for confirmation. The syntax for parameters is similar to Pascal's syntax except for the following two cases: 1. Var parameters must be prefixed with a '↑'. 2. Normal Pascal constants are treated as integer decimals if they are in the range -32768..32767, otherwise as long decimals. Other constants have to be prefixed by a '#', a base indicator and a type indicator as described in section 2.1. For example:

```
Execute Foo(↑Bar, 34, #H5FA4, #BL0111, Count)
```

calls procedure **Foo** with reference parameter **Bar**, decimal integer 34, hexadecimal integer 5FA4, binary long 0111 and value parameter **Count**. The variables **Bar** and **Count** are visible from the current point of execution.

If R is a *function*, more restrictions have to be observed, most of them due to the insufficient symbol table information provided by the compiler: Functions names must always be indicated by parentheses followed by a Type Qualifier. Parentheses are needed even if the function does not have any parameters, otherwise the result of the function is denoted. Functions can be used in arbitrary expressions, but cannot be passed as parameters.

WARNING: *It is easy to do something wrong when calling a function from inside the debugger. The following examples show how to use function calls and how not:*

```
Type REC = record i : integer; l : long end;
Var b : boolean; r : REC;
Function foo(i : integer): boolean;
Function fool: boolean;
Function bla(s  : string[80]): REC;
```

Legal calls:

```
b := foo(i):boolean
b := fool()
fool()    -- displays result of function call on screen.
r := bla('sdfsfd':string[80]):record[3] -- assigns the value of bla to r
bla('sdfsfd':string[80]):record[3] --types the value of bla on on the display.
```

Illegal calls:

```
b := foo(i)      -- missing type qualifier
```

Problematic calls:

```
b := foo1        -- Missing parentheses:
                    The debugger takes the current return value
                    of foo1 instead of calling foo1.
r := bla('sdfsfd'):record[3]  -- takes default value of string length.
```

Go

Resume the execution of the suspended target process.

Halt

Suspend the execution of the target process.

Run T

Delete the link to the current target process and start the execution of target process T. This command is useful for debugging a new target process, for example if the current program has been recompiled and relinked, without leaving the debugger. NOTE: If you just want to restart the current target process, use the **Execute** command with the name of the main program as parameter.

TerminateTarget YES|NO

Terminate or do not terminate the target process on QUIT. Initially **TerminateTarget** is set to **YES**.

Unwind

Remove all activation records from the runtimestack up to and including the last routine that was called with the EXECUTE command. If there is no such routine the stack is not changed.

4.11 Miscellaneous Commands

@F

Execute command file F. If F is not specified, the command file M.kmd is executed, where M is the extensionless file name of the main program[10]. Command files can be nested to an arbitrary depth. If F does not contain an extension and there is no file with the name F, then the extension .kmd is appended.

---

[10] M.kmd is the default command file. If it can be found with the initial file search list, it is automatically executed at the beginning of the debugging session.

$M

Execute the MACE command M without leaving the KRAUT command interpreter. MACE commands are described in section 5.

--

Start of a comment. Any string following the two dashes up to the end of the line is regarded as comment.

&

If the result of the last debugging command was a numeric value, it can be denoted by the symbol **&**. **&** can be used in any Pascal expression. The symbol &↑ interprets the value of the last command as a pointer. Thus it especially useful for pointer chains.

Address ID

Show the target address of ID. ID denotes an identifier as defined in section 2.3. If ID is a routine, then show the addresses of its entry and exit points. If ID is omitted, the identifier from the last debugger command is taken. The address command allows you to inspect ID's more closely with MACE if KRAUT does not provide any help (for example for record fields).}

Compute E

Compute expression E. The **Compute** command evaluates (nearly) any pascal expression. The keyword COMPUTE can be omitted if the expression starts with a constant [11]. If the type of a variable is not known or has to be recasted, use type qualifiers. For example, if I is an integer and W a function of type integer and the result is of type real, the following command computes the real value 400022 + 5*7-i+w(i):

```
COMPUTE (400022 + 5*7-i+w(i):integer):real
```

Current M

If M is a module, set the *current module* to M (This is equivalent to **Open** M). Otherwise, if M is an active routine, set the *current routine* to M. If M is not active, do not change the *current routine*. If no argument is specified the *current module and routine* are displayed.

---

11 Note that any *single* digit will be interpreted as a source line number of the TYPE command.

Exception

Show the current exception.

Help

Creates a help window and allows the user to interactively peruse the appendix of this manual.

Indent N

Concatenate `N` blanks with the prompt sign. NOTE: In the case of a command error or when `↑DEL c`, `↑DEL k` or `↑DEL d` is typed, `INDENT 0` is automatically executed.

Maintainer M

Print out the name of the maintainer of module `M`. This command prints out the string following the key word **Maintainer** in the comment switch in module M. For example, if module TEST contains `{$Comment Maintainer bob@cmux x3828}` in the source file, then "`bob@cmux x3828`" is the maintainer of TEST. If M is omitted in the command, then the maintainers of all maintained modules are printed out.

Mace

Enter the command interpreter of MACE, the low level Accent debugger. MACE commands are described in section 5. To return back to the KRAUT command interpreter type the MACE quit command.

News

Print out news about undocumented features, new bugs and fixes of old bugs.

Quit

There are four possible courses of action when choosing to quit the debugger. They are as follows:

Quit	Exits the debugger.
Continue	Resumes the debugging interaction.
Report	Refer to the Report Command below.
SaveScriptAndQuit	Automatically saves a transcript of the current dialogue if the transcript switch was **On** at least once during the debugging session;then exits from the debugger.

Report M

Create the transcript of the debugging session to be deposited in the outgoing mail for the mailing address M when the QUIT command is typed. If M is omitted, the account `Spice@spice` is assumed as the receiver. When quitting you are asked to confirm the mailing address, the FROM field (which is taken from the first line of the file <boot>sysname) and whether you want to provide an additional message. The message can be in a file or it can be typed in the window. A typed message has to be terminated with a single dot on a line. The **Report** command can be typed at any time of the debugging session. Note that KRAUT is not doing the actual mailing. It only puts the transcript in a file **<boot>perqout.R***, from where it has to be picked up by the mail system.

EXAMPLE (user input is underlined):

```
|Report bob, dzg
|Mail request to bob, dzg registered
| ...
|Quit
|Action (Quit, Continue, Report, SaveScriptAndQuit) [REPORT] ? <CR>
|Mailing  transcript...
|TO: [bob, dzg] bob@cmu-cs-spice, hibbard@cmua, ets
|FROM [bob@cmu-cs-spice]: <CR>
|Subject [Bug in test]: <CR>
|Do you want to add an initial message? (Yes, No) [YES] Yes
|Enter file name or hit <RETURN>: <CR>
|Type message (terminate with a "." alone on a line):
Hi, I found the following bug:
.
|Preparing mail...
| ...Mail deposited for bob@cmu-cs-spice, hibbard@cmua, ets
```

Script on|off

Turn the transcript `on` or `off`. If `Script off` is contained in the default command file, no transcript file will be generated at all.

Silence on|off

If silence is on, then the commands executed in a command file will not be echoed on the curent window. If silence is off, they are echoed. The default value is off.

Shell

Spawn another shell, inheriting the environment in which KRAUT is currently running.

Show P

Depending on the parameter P the following is displayed:

P = Breaks	Show the current break point list.
P = Counters	Show the currently defined counter variables.
P = Models	Show the current models.
P = Modules M	If M is omitted, then type KRAUT'S module list. Otherwise give some information about module M. This includes date of compilation, names of seg, qmap and sym files, name of imports, etc.
P = Pathfunctions M	Show the user defined path functions for module M. If M is omitted show the user defined path functions of all modules.
P = Pathrules P1	If P1 = 'active' then display the list of enabled pathrules: if P1 = 'inactive', then display the list of disabled path rules, otherwise display all path rules currently defined.
P = Routines M	Show the routines and user defined path functions for module M. M must be in the module search list. If M is omitted show the routines and user defined path functions of all modules.
P = RunFile (R M)	If R and M are omitted, print the core image of the current target process into the file M.map, where M is the extensionless filename of the main program. If R and M are given, then print the contents of run file R into file M. Note that in this case some fields in the file and segment entry descriptors are set to 0 and do not describe locations in the address space!
P = SearchList	Display Accent's file search list.
P = Window	Display the currently defined windows.

Version

Type the version number of the debugger.

4.12 Window Commands

The following commands allow the user to define typescript windows, change from one window to another and display information in any defined window. There are two predefined windows, which are treated differently from user defined windows: The `DEBUG` window is the window allocated by Accent when the debugger is started up. The `DEBUG` window is treated in such a way that it allows the debugger to run even if the window manager is not running. The `BargraphWindow` is a graphic window in which variables can be bound to bargraphs (see section 4.13).

There is always a *current window*, which is originally set to DEBUG. There may be a maximum of ten windows, each of which may be manipulated by all of the normal SAPPHIRE commands (top, bottom, move, grow, etc.).

Window W LX TY W H

Creates window W with upper-left corner LX, TY and W characters per line and H lines. If these coordinates are omitted, default coordinates are used.

Window W

Creates window W. Typing **Window W** a second time brings window W to the top and makes it the listener. This command is used to move the listener from one window to another.

DWindow W

Deletes the window with the name W.

DWindow *

Deletes all windows but the current one.

C >> W

Redirect the output of the KRAUT command or command sequence C into window W but do not leave the current window. If W has not yet been defined, a window with default coordinates will be created.

4.13 Model Commands

Model commands allow you to display variables with a format different from the built-in display format. A *model* is a display template defined by a **Model** command. Bargraphs are predefined models. Models are instantiated by the **Bind** command and it is possible to bind several variables to the same model.

Model Bargraph B MAX S

Create or rescale a bargraph with the name B, where B can be any identifier. If B has not yet been defined, a template for B is created. The maximal value of the template is given by the integer MAX. S describes the scale and can be either LINEAR (*linear* scale) or LOG (*logarithmic* scale). MAX and S can be omitted in which case the defaults MAX = 32767 and S = LINEAR are assumed. If the bargraph B already exists, the template and all its instantiations are rescaled according to the new values MAX and S. Note that the keyword Model can be omitted in the command.

Model Routine M R

Create a model with the name M, where M can be any identifier, and bind the routine R to it. The keyword `Model` can be omitted in the command. R can be any procedure defined in the user program and may have parameters. All except for one of R's parameters have to be bound at the definition of the model. It is possible to mark one of the parameter's slots with a "$" character and this parameter will be bound by the **Bind** command. For example,

```
Model Routine M Foo(↑gi, #123, $)
```

binds the routine `Foo` with `gi`, `#123` and $ to the model M, and $ will be provided by the **Bind** command.

Bind ID M

Bind the variable `ID` to a model with the name `M`. Currently two classes of bindings are possible:

1. If `M` is a bargraph, then `ID` can be a counter, a history variable (ACT or TERM) or a variable of type integer. The debugger allocates a slot in the window `BargraphWindow` and displays ID according to the attributes of M[12]. For example

   ```
   Bind ←ACT(Foo) M
   ```

   binds the history variable `←ACT(Foo)` to the bargraph `M`. After a variable is bound to a bargraph, any *display* command will display its value according to the scale and maximum value of the bargraph. If ID is a counter or a history variable, the bargraph is updated on any *assignment*. However, if ID is a program variable, the bargraph is not updated until the next *display* command is issued.

2. If `M` is a routine, then `ID` can be a variable of any type. If `M` has been defined with parameter "$", then `ID` is substituted for "$" in `M`. For example, the commands

   ```
   Model Routine M Foo(↑gi, #123, ↑$)
   Bind grec M
   ```

   define a model M for routine `Foo` and bind `grec` to it. Displaying `grec` will result in calling `Foo(↑gi,#123, ↑grec)`.

---

12 If `BargraphWindow` does not yet exist, you are prompted for the screen locations.

DModel M

Delete the model M.

DBind ID

Delete ID's binding to a model and rebind it to text format.

Chapter Five
MACE

The command syntax for MACE is quite different from that for KRAUT. Some MACE commands may contain MACE **expressions** and qualifiers to change the radix and mode of the entities being displayed. The radix or mode qualifier is optional and if omitted the current radix or mode is assumed, respectively. The radix qualifier controls the radix of numbers and can have one of the following values:

#	Octal radix
.	Decimal radix

The mode qualifier specifies the type of object(s) to be displayed and can have one of the following values:

a	16 bit integer, 32 bit long, byte, character and string (max 15 characters)
b	Byte
c	Character
i	16 bit integer
l	32 bit long
m	IPC Message
q	Qcode
r	Floating point
s	Pascal string

A MACE **expression** is a parenthesized expression of primary values using one or more of the following operators:

↑	Dereference (written after the value)	6
-	Negation	5
~	Logical inversion	4
*	Multiplication	3
/	Division	3
!	MOD	3
+	Addition	2
-	Subtraction	2
&	Logical And	1
%	Logical Or	1
<	Leftshift	1
>	RightShift	1

The operators are ordered in decreasing precedence. Parentheses can be used to indicate different precedences.

A **primary value** is one of the following constructs, where <M>, <P>, <N> and <E> are MACE expressions denoting addresses and numbers.

892	Any decimal constant
#377	Any octal constant
.	The last location referenced.
<M>	Base address of the code segment of module M.
<P>	Entry point of routine P.
<M>.*G*<E>	Offset E in the global frame of module M.
<M>.*R*<E>	The E'th word in the routine dictionary of module M.
<M>"<P>	Entry point of routine P in module M.
<P>.*Q*<E>	Offset E in routine P.
<P>.*A*<E>	The E'th word of the ACB for last call to P.
<P>.<N>*A*<E>	The E'th word of the ACB for the N'th call to P.
<P>.*F*<E>	The E'th local word of last call to P.
<P>.*R*<E>	The E'th word in P's routine dict entry.

Examples for MACE expressions are

Test3.Proc1 + #557↑	Procedure *Proc1* in module *Test3* plus contents of address octal *557*.
Proc1.4A3	The *third* word in the ACB for the *fourth* call of procedure *Proc1*.
Foo.Q23 - .	Qcode offset *23* in routine *Foo* minus last expression referred to.

The following example shows the order of evaluation of MACE expressions:

```
Foo.Q5 * Test.G45 & ~3 - 5↑
```

is evaluated as

```
(Foo.Q5 * Test.G45) & ((~3) - (5↑)).
```

5.1 MACE Commands

In the following <E> and <V> denote MACE expressions, <R> a radix qualifier, <M> a mode qualifier and <D> a decimal integer. Note that command names are case sensitive, whereas MACE expressions are not.

?, h, H,'?

Type help information. The commands ?,h and H explain MACE expressions and the ";", "=" and ":=" commands. The command '? explains the "'" commands.

<E>; <R> <M> <D>

Display <D> target memory locations starting at <E> in the format specified by mode <M> and radix <R> in units appropriate for the given type. <D> must be a decimal integer. <E> can be any MACE expression. For example:

```
Foo.q54;.q23
```

displays 23 locations starting at address `Foo.q54` as qcode instructions with decimal arguments.

<E>=<R> <M>

Calculate the MACE expression <E> in mode <M> using radix <R>. For example:

```
87506 + 45*7=#i
```

displays the value of `87506 + 45*7` as an octal integer.

<E>:= <V>

Store the value <V> in the 16 bit location <E>. <E> and <V> can be arbitrary MACE expressions. For example:

```
Test.G456 := Foo.F4
```

stores the value of the 4'th local word of routine `Foo` into offset 456 in module `Test`.

<D>'a

If <D> = 1,2...,100, then show the contents of the <D>'th activation record (ACB) on the stack. <D> = 1 means the ACB of the top routine and <D> = n means the n'th ACB down towards the bottom of the stack. If <D> is larger than 100, MACE assumes you have typed the address of the ACB[13]. The ACB consists of the following information:

```
ACBSL:          Activation pointer (AP) of enclosing routine
ACBLP:          Local pointer (LP) of this routine
ACBDL:          Activation pointer of calling routine
ACBGL:          Global pointer (GP) of calling routine
ACBTL:          Top pointer (TP) of calling routine
ACBRS:          System Segment number (SSN) of calling routine
ACBRA:          Program counter (VPC) of calling routine
ACBRR:          Routine Number of calling routine
ACBEP:          Pointer to current exception enable record
ACBStackSize:   Size of EStack of calling routine
ACBStackSize+i: Saved EStack values ,i= 1,...,ACBStackSize
```

<E>'b

Set a breakpoint at address <E>. If <E>=0, show the list of current breakpoints.

<E>'d

Delete a breakpoint at address <E>.

'e

Display expression stack (from the bottom towards the top).

<D>'E<V>

If <D>= -1, then assign <V> to the micro state register EStkCount, which holds the current size of the EStack. If <D> in [0..15] then assign <V> to expression stack register <D>. (Note: 0 is the bottom of the expression stack).

<E>'h

Display the history for message <E>. If <E>=0 then display the history of all messages. This command assumes that the program you are debugging imports the module `AccCall from AccCall.pas`. If this is not the case, you get the error message `No module specified.AccCall` must be specially compiled to enable the message history mechanism. (The message history mechanism is currently disabled.)

[13] The activation pointer (AP) of a routine points to the address of its ACB. See 'm command.

<D>'m

If <D>=0, show the qcode state of the *current routine*[14]. Otherwise show the qcode state for the <D>'th activated routine on the stack, counting downward towards the bottom of the stack. Thus, <D>= 1 means the routine on top of the stack, etc. The *qcode state* consists of the following information:

```
SB:  Stack base
CB:  Base address of code segment
GP:  Global pointer
VPC: Program counter
RN:  Routine number
LP:  Local pointer
AP:  Activation pointer   (Base address of ACB)
TP:  Top pointer
```

'M

Displays the contents of registers 5 to 16 and 110 of Accent's micro context block. A typical output for the 'M command looks as follows:

```
Program counter            [R 5]: 172[#254]
Top pointer                [R 6]: 21290[#51452]
Activation Pointer         [R 7]: 21280[#51440]
Global Pointer             [R 8]: 2562[#5002]
Local Pointer              [R 9]: 21236[#51364]
Routine Number             [R 10]: 0[#0]
Code Segment               [R 11]: 152[#230]
Stack Segment              [R 12]: 1[#1]
Breakpoint register        [R 13]: -1[#-1]
Exc handler code segment   [R 14]: 152[#230]
Exc handler global pointer [R 15]: 2562[#5002]
Encode overlay # and entry [R 16]: 3[#3]
VM status                  [R 110]: 1280[#2400]
```

<E>'p

If <E> is an address within the code of any routine, then display the routine dictionary of that routine. The routine dictionary has the following format:

```
RDPS:     Size of parameters
RDRPS:    Size of result + parameters
RDLTS:    Size of locals + temporaries
RDLL:     Lexcial Level
RDEntry:  Entry address relative to segment
RDExit:   Exit address relative to segment
```

14 The current routine can be changed with the KRAUT runtime stack commands `UpStack`, `DownStack`, etc.

<E>'P

If <E>=0, then show the code segment base address (CB) and the global pointer (GP) of each module. Otherwise, show the segment header block for module <E>. The time stamp attached to a file name indicates the last time the file was updated. For example, the command `test3'P` might display the following information:

```
Program TEST3 [Compiled at: 11 Apr 83 11:33:29]
Src file  = <boot>user>bob>test3.PAS [20 Aug 82 10:53:01]
QVersion  = 3
GDB size  = 184 words
Version   = ''
Comment   = ''
QMapFile  = <boot>bob>test3.QMAP [11 Apr 83 11:33:29]
SymFile   = <boot>user>bob>test3.SYM [11 Apr 83 11:33:29]
Imports 3 files: (Import info at block 2)
        TEST3A from test3a.PAS
        WRITER from WRITER.PAS
        STREAM from STREAM.PAS
```

'q

Terminate MACE and return to KRAUT command level. Note: This command should only be executed on MACE command level.

'r

Resume execution of the target process.

's

Suspend execution of the target process.

<D>'t

Show the last <D> routine calls starting at the top of the stack.

<D>'u

Show the contents of micro state register <D>.

<D>'U <V>

Assign the value <V> to micro state register <D>.

'x

Toggle MACE debug flag (for debugging purposes only).

'X

Toggle Expression trace flag (for debugging purposes only).

'@

Low, low level debugging aids (type '@? for help if you are really desperate).

Chapter Six
Coping with Debugger Bugs

KRAUT is still actively being developed and an internal error might show up while you are debugging your own program. If the error is an exception defined in "`except.dfs`", the debugger writes out an explanation such as DIVISION BY ZERO, otherwise the exception is written out as a segment and routine number-pair. You have one of the following options:

- RECOVER (Default): Try to return to command level.
- DEBUG: Recursive debugging mode. You are prompted for a new window to debug the debugger.
- QUIT: Quit the session.

The following command is intended for debugging the debugger.

Verbose

> Set debugging switches interactively. The switches can also be set initially by a file M.switches, where M is the name of the file containing the main program.

If you run across an uncaught exception in the debugger, it would be helpful if you could do the following:

- Recover from the bug.
- Turn on the switches with the **Verbose** command.
- Repeat the command that caused the exception.
- Enter the recursive debugging mode and print the call stack of the current state of the debugger.
- Save both transcript files and send them to bob@CS-SPICE (Use the **Maintainer** and **Report** commands).

Chapter Seven
Current Restrictions and Known Bugs

- It is not possible to **Open** modules which are not imported by one of the modules of the program.
- Call chains are not yet implemented [15].
- Consistency checks between source and object files are not implemented.
- The predicate ←UNDEFINED (see page 5) is not implemented.
- The general notation `M''P1'...'Pn-1'Pn`, `i` denoting variable `i` in routine P_n nested in P_{n-1},..., nested in P_1, which is declared in module `M` has not yet been implemented. Only `P'Foo` to denote the variable `Foo` in routine `P` and `M''P'Foo` to denote the variable `Foo` in routine `P` in module `M` are implemented.
- Because of restrictions in the symbol table, it is not possible to access record fields by name. Word (= 16 bit) offsets relative to the base address of the record must be used instead. To determine word offsets, you have to know how the compiler allocates storage for record fields. For example, in the following record definition

```
Var   Rec1 : record
               Rec2: record
                       i : integer;
                       Rec3: record
                                p : long;
                                foo: ↑integer;
                             end;
                     end;
               b: boolean;
             end;
```

  the field `Rec1.Rec2.Rec3.foo↑` is denoted by the expression `Rec1.0.1.2↑`, where `Rec1.0` is the offset of field `Rec2` in record `Rec1`, etc. Note that the compiler allocates field list identifiers separated by commas in the reverse order in which they are mentioned in the declaration. For example, in the following definition

```
Var   Rec1 : record
              i,j,k,l,m,n : integer;
             end;
```

  the expression `Rec1.0` denotes the field `Rec1.n` and `Rec1.4` denotes `Rec1.k`. The above examples are only valid for unpacked records. Ask a compiler hacker if you want to examine packed records!

- Array elements are not fully supported: Because there is no symbolic information available for the bounds of arrays, the debugger assumes 0 as the lower bound for array variables. Thus for arrays starting on a different lower bound, the lower bound has to be normalized to 0. Multidimensional arrays are not supported.

---

[15] However, one can access allocated but unvisible variables by positioning with **TopStack**, **BottomStack**, **DownStack** and **UpStack** followed by **Locals**.

- Named Constants (defined with CONST) are not supported.
- Display of Records: Records are displayed on a rather low level. KRAUT calls MACE to successively interpret each word of the structure as an integer, a long (using two words) , two bytes, two characters or the start of a string. The number of words to be displayed can be specified or a default value is assumed: For records the number of words necessary to store the record and for arrays the size of the subcomponent type. For example, given the program fragment

```
Var   grec : record
                int: integer;
                bool: boolean;
      end;
....
grec.int:= 11111;
grec.bool := true;
```

grec can be displayed as follows:

```
|grec
|Record!    [TEST3]
|Display how many words? [2]1
|            Integer         Long  LSB MSB   Char       String[15]
|TEST3.G172:   11111.       76647. 103. 43. | g +| [103]"+##########R#R##"
```

The next command (<CR>) can be used to display successive words:

```
|<CR>
|TEST3.G173:       1.      131073.   1.  0. |↑A↑@|   [1]"#"
|<CR>
|TEST3.G174:     255.       65791. 255.  0. |XX↑@| [255]"##########R#R##"
```

The pair XX is used to indicate a byte which cannot be interpreted as an Ascii character.

- Because there is no symbolic information for enumerated types, sets are displayed as bit sequences. The leftmost bit refers to the first literal, and the n'th bit from the left to the n'th literal of the set. For example, given the program fragment

```
type
  color = (red, green,brown,yellow,black,pink);
var
  cset: set of color;
....
 cset := [red,green,pink];
```

the variable cset will be displayed as

```
|cset
|TEST3.G200:  (bits 0..15)  1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0
```

- User defined enumerated types must be denoted by their integer equivalent.
- String search does not work correctly for files that contain one or more INCLUDE files.

Appendix A
Kraut Command Summary

Breakpoints. Page 12

```
Break [after] R|#     : Set breakpoint at [after] routine R or line #.
DBreak R|#            : Delete breakpoint at routine R or line #.
DBreak *              : Delete all breakpoints.
```

Constant syntax. Page 4

Any Pascal constant is a valid constant. DECIMAL Pascal constants in the range -32768.. + 32767 are of type integer (16 bits), otherwise of type long (32 bits). Constants in a different radix have to be prefixed by a #, a radix indicator (B,D,O,H) and a size indicator (I,L). (B = binary,D = decmial,O = Octal,H = hexadecimal, I = integer,L = long). The default radix is D, the default size I.

Examples:

```
#OL4712347      32 bit octal
#LHABFD4CD9     32 bit hexadecimal
#b10101011      16 bit binary (default: I)
#bL10101011     32 bit binary
3434            16 bit decimal
9896966         32 bit decimal
```

Declarations. Page 14

```
Counter C (:= I)   : Define Counter C and initialize it to 0 (to I).
DCounter C         : Delete Counter C.
Define Function F  : Define path function F.
```

Editor commands. Page 13

```
Around (i F)       : Show 20 source lines around current line.
                     (around line i in file F).
Grep 'S' (F)       : Look for all occurrences of string S.
                     starting at current line (starting at line 1 in F).
Scroll (i F)       : Show next 20 source lines from current line
                     (from line i in file F).
(Type) i (j F)     : Show source lines  i (to j in file F).
'S'(!) F           : Search forward in file F for string S.
```

Identifier syntax. Page 4

```
Foo                Identifier Foo  (following Pascal's scope rules)
Proc'Foo           Identifier Foo in routine Proc
Mod''Proc'Foo      Identifier Foo in routine Proc in module Mod
Foo:T              Coerce identifier Foo into type T, where T can be
                   any type qualifier.
```

Inspection commands. Page 17

```
&              : Value of last command.
Address (ID)   : Get address of last variable (of ID).
Compute E      : Compute expression E.
Current R      : Move down in the run time stack to R and make it the
                 current routine.
Globals (M)    : Show all glohals of current module (of module M).
Locals (R)     : Show all variables of current routine (of routine R).
Parameters (R) : Show all parameters of current routine (of routine R).
Radix N        : Set the current radix to N where N is a DECIMAL (!)
                 number in { 2,8,10,16}). Initially the current radix is 10.
V(=)           : Type the value of V ('=' is needed, if V is also a command).
VAR:= ID       : Assign the value of ID to VAR.
Write(p1,...,pn),
Writeln(p1,...,pn): Write the pascal expressions p1,...,pn (and CRLF).
```

Line number syntax.

```
.                          Current line in current source file.
24                         Line 24 in current file.
24 test3                   Line 24 in file test3.pas;1.
24 test3.pas;2             Line 24 in file test3.pas;2.
24 sys>accent>test3        Line 24 in file sys>accent>test3.pas;1.
```

Mace. Page 29

Type MACE to enter the Mace interpreter. Type '? to Mace for further help. Mace commands can also be typed to Kraut: Type $M to the Kraut interpreter to execute the mace command M.

Miscellaneous. Page 21

```
@FILE.kmd          : Execute default command file (FILE.kmd).
$M                 : Execute Mace command M.
--                 : Start of a comment.
Current M          : Set current module to M (same as OPEN M).
Exception          : Show the current exception.
Expert (ON|OFF)    : Expert switch. (Default: OFF).
Indent N           : Concatenate N blanks with the prompt sign.
Shell              : Create another shell with current environment.
```

Model commands. Page 26

```
(Model) Bargraph B MX S    : Define (or rescale) bargraph B with maximum
                             value MX and scale S (Linear or Log).
(Model) Routine M R        : Define a model M and bind the routine R to it.
Bind ID M                  : Bind variable ID to model M.
DModel M                   : Delete model M.
DBind ID                   : Delete ID's binding to a model.
```

Next command. Page 18

```
<CR>          : Execute the previous command with new arguments.
                Implemented for $, Around, Calls, DownStack, Search,
                SingleStep, Type, UpStack.
```

Pathrule Commands. Page 14

The following commands are defined for path rules. If only the name R of the path rule is typed, you are prompted for the missing arguments:

```
Action R A                  : Bind action A to path rule R.
CheckPath R G               : Qualify G as a CHECK path rule with name R.
Disable R                   : Enable path rule with name R.
EditPathRule R              : Edit path rule R.
Enable R                    : Disable path rule with name R.
FindPath R G                : Qualify G as a FIND path rule with name R.
PathRules (active|inactive): Show (active or inactive) path rules.
PathRules verbose           : Verbose while interpreting path actions.
Remove (Action) R           : Delete path rule R (or action only).
```

Reporting bugs.

```
Maintainer (M)  : Get name of maintainer of all modules (of module M).
News            : Print any news about new features, known bugs, etc that
                  are not undocumented in the Spice manual.
Report A@M      : When QUITting the debugger, the script file will be
                  given to the Spice Mail system to mail it to address A at
                  machine M.
Script ON|OFF   : Set transcript switch ON or OFF (default: ON).
Verbose         : Set various debugging switches (For internal debugging).
Version         : Get the current version of KRAUT.
```

Runtimestack commands. Page 18

```
BottomStack          : Move to  bottom of stack.
Calls (#)            : Show current call sequence(# levels).
DownStack (#)        : Move backward 1 (#) routine(s) in dynamic call chain.
TopStack             : Move to top of stack.
Unwind               : Remove last EXECUTED routine from stack.
UpStack (#)          : Move forward 1 (#) routine(s) in dynamic call chain.
```

Show command.

```
Show P            : P can be one of the following values:

     P = Breaks          : Show the current break point list.
     P = Counters        : Show the currently defined counter variables.
     P = Models          : Show the current models.
     P = Modules         : If M is omitted, then type KRAUT's module list.
                           Otherwise give some information about module M.
                           This includes date of compilation, names of seg,
                           qmap and sym files, names of imports, etc.
     P = Pathfunctions M: Show the user defined path functions for module M.
                           If M is omitted, show them for all modules.
     P = PathRules P1    : If P1='active', then display the list of enabled
                           pathrules, if P1='inactive', then display the list
                           of disabled path rules, other display all currently
                           defined path rules.
     P = Routines M      : Show the routines and user defined path functions
                           for module M. If M is omitted then show them for
                           all modules.
     P = Runfile R M     : Dump the contents of run file R into file M.
```

```
P = Searchlist      : Display Accent's file search list.
P = Windows         : Display the currently defined windows.
```

Searchlist commands. Page 10

```
ChDir D             : Set current directory to D. If D is missing,
                      show current directory.
Close M             : Close module M ( * for all modules).
Open M              : Open module M  ( * for all modules).
SetSearch L         : Modify file search list  : - (pop), ID (push).
System              : Open PascalInit and all its imports.
```

Target Process commands. Page 19

If the target process is RUNNING, the following commands and control characters can be typed. Note that control characters have to be typed in the window of the debugger, not that of the target process:

```
↑DEL t                          : Show the state of the target process.
                                  (Not yet implemented)
↑DEL c, ↑DEL k, ↑DEL d          : Return to command level.
Halt                            : Suspend execution of target process.
```

If the target process has STOPPED the following commands can be executed:

```
P(p1,...pn),
Execute P(p1,...pn)     : Execute routine R with parameters p1,...,pn. R can
                          also be the main program.  Omit closing ')' for
                          parameter check.
Go                      : Resume execution of target process.
Run (A)                 : Start execution of target process (with arguments A).
TerminateTarget YES|NO  : Do or do not terminate target process on QUIT.
```

Trace commands. Page 11

```
Trace [Before|After] R [C] : Trace routine R (only its entry or exit)
                             and perform commands C. If R is a module
                             set traces for all of the routines of R.

DTrace [Before|After] R    : Delete trace of R (only entry or exit).
                             If R is a module, delete all traces for
                             module R.
```

Type qualifier. Page 4

```
Array[n]   array (n words)  |  Pointer        pointer
Boolean    byte (8 bit)     |  Record[n]      record (n words)
Char       char             |  REal           real
Integer    integer (16 bit) |  String         string
Long       long (32 bit)    |  SEt[n]         set    (n words)
```

Window commands. Page 25

```
Window W LX TY W H   : Create window W with upper left corner LX,TY and
                       W characters per line and H lines.
Window W             : Move to W if it exists, otherwise create window
                       with default coordinates.
DWindow W            : Delete window W.
DWindow *            : Delete all existing debug windows except for the
                       current one.
C >> W               : Redirect output for command C into window W.
                       Window W is created with default coordinates,
                       if it does not yet exist.
```